VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Procesor relační algebry

Relational algebra processor

2015

Jiří Buchlovský

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jiří Buchlovský**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Procesor relační algebry
Relational Algebra Processor

Zásady pro vypracování:

Cílem práce je navrhnout a naimplementovat paměťově orientovaný procesor relační algebry s využitím dostupných datových struktur zvolené vývojové platformy.

Práce bude splňovat následující body:
1. Rozbor relační algebry a popis standardních relačních operátorů.
2. Návrh architektury procesoru.
3. Implementace jednotlivých operátorů.
4. Návrh dotazovacího jazyka využívajícího naimplementované operátory.
5. Implementace navrženého jazyka, testování a porovnání s existujícími procesory.

Práce může být eventuálně rozšířena o zpracování jednoduchých dotazů SQL nebo o vizualizaci operátorových stromů.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Petr Lukáš**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 16. dubna 2015 .....

Rád bych na tomto místě poděkovala všem, kteří mi s prací pomohli, protože bez nich by nevznikla, především však mému vedoucímu Ing. Petru Lukášovi za pomoc a odborné rady, díky kterým jsem byl schopen tuto práci úspěšně dokončit.

Abstrakt

Cílem této práce je navrhnout a implementovat procesor relační algebry, který bude vykonávat příkazy nad daty a zároveň vytvoří plán spouštění. S tím souvisí návrh jazyka a implementace jeho operátorů. Také je třeba představit relační algebru, čímž se bude zabývat úvod práce a porovnáním programu s existujícími procesory v závěru práce.

Klíčová slova: relační, algebra, operátory, procesor, databáze, jazyk, množina, implementace, strom, data

Abstract

Main goal of this thesis is design and implementation of relational algebra processor capable of executing queries over actual data and creating execution plan, which will be stored in tree structure. Another thing associated with processor is design of language and implementation of its operators. Another very important topic is relation algebra itself, which will be described at the first part of the thesis. Last part of thesis is to compare my processor with already existing programs.

Keywords: relational, algebra, operators, processor, database, language, set, implementation, tree, data

Seznam použitých zkratek a symbolů

SQL	–	Sequence Query Language
CSV	–	Comma Separated Values
EOF	–	End Of File

Obsah

Seznam tabulek

Seznam obrázků

Seznam algoritmů

1 Úvod

1.1 Cíl práce

Ukládání dat v podobě relací se postupem času ukázalo jako velice efektivní vzhledem ke snadné implementaci a efektivnímu dotazování. Cílem práce je popsat relační algebru a její operátory společně s návrhem a vytvořením procesoru, který bude provádět příkazy nad daty. Dále je důležité program otestovat a porovnat s podobnými existujícími procesory.

1.2 Rozsah práce

V úvodu práce se budeme věnovat rozboru relační algebry a popisu standardních relačních operátorů. Následuje návrh implementace architektury procesoru, na který navazuje implementace jednotlivých operátorů. Dále se budeme věnovat návrhu dotazovacího jazyka využívající implementované operátory. K tomu se váže část zabývající se implementací navrženého jazyka.Závěr práce se zabývá testováním vytvořeného programu, porovnání s existujícími procesory a shrnutí.

2 Relační algebra

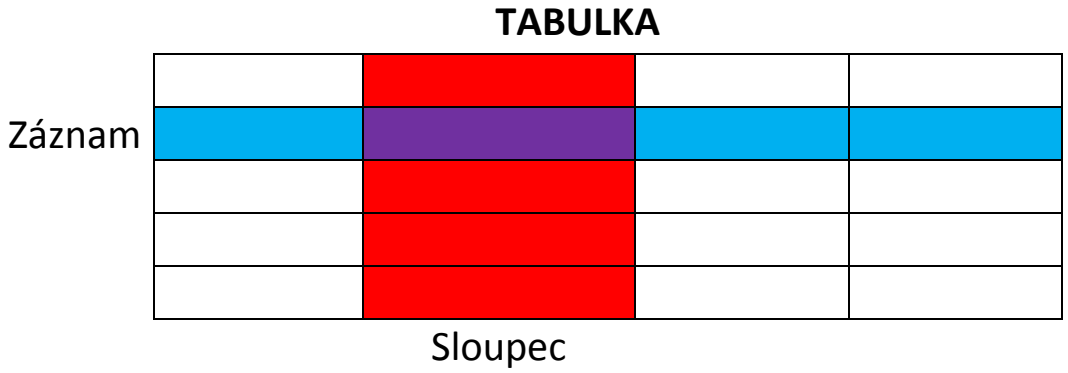
2.1 Úvod do relační algebry

Algebra je odvětví matematiky, které se zabývá abstraktností pojmů a vlastností elementárních matematických objektů. Dělí se na algebru *elementární* a *abstraktní* (moderní). Elementární algebra se zabývá symbolickou manipulací s výrazy a řešením rovnic. Abstraktní algebra studuje obecné algebraické struktury. Algebra má vždy definovanou množinu M a operátory. *Operátor* je funkce (zobrazení) z kartézského součinu M $\times$ M $\times$ ... $\times$ M do množiny N. Podle velikosti kartézského součinu určujeme *aritu* operátoru (unární, binární, atd.) [1].

Relační algebra je typ algebry sloužící dotazování dat uchovaných v podobě relačních databází. V relačních databázích jsou data uchovávána v tzv. relacích. *Relace* je konečná množina n-tic $d_1, d_2, ..., d_n)$ kde každý prvek d_j je členem domény D_j. *Doména* představuje všechny možné hodnoty, které může data element obsahovat. Relaci je možno definovat také jako konečnou podmnožinu kartézského součinu těchto domén.

Pro dotazování nad relacemi se nejčastěji využívá jazyk SQL. Jde o deklarativní jazyk, ve kterém uvádíme, co chceme na výstupu, ale neudáváme přesný algoritmus. Naproti tomu, relační algebra je jazyk, ve kterém přesně specifikujeme, jak bude výsledek vypočten. Úkolem každého procesoru SQL dotazů je nejprve přeložit SQL dotaz do nějaké podoby relační algebry. Obvykle se nepoužívá relační algebra podle výše uvedené definice, ale je podle potřeby upravena.

V našem případě je relační algebra tak, že nepracuje s relacemi jakožto množinami n-tic, kde je automaticky vyloučena duplicita, ale bude pracovat s tabulkami, ve kterých se n-tice opakovat mohou. Z těchto důvodů budeme nadále používat pojmy tabulka a záznam.



Obrázek 1: Příklad tabulky

Na Obrázku č. 1 jde vidět strukturu tabulky. *Tabulka* je uspořádaná multimnožina záznamů. Je rozdělená na *Sloupce* a *Záznam* odpovídá řádku v tabulce.

2.2 Operátory relační algebry

V této kapitole uvedeme základní operátory relační algebry. Mezi základní operátory, které zmínil E.F. Codd ve své publikaci o relační algebře[2], patří projekce, selekce a množinové operace – sjednocení, rozdíl a kartézský součin. Existují ale další relační operátory jako průnik, přejmenování a spojení, kterým se v této práci budeme věnovat také. Definice uváděné v této kapitole jsou přizpůsobeny potřebám této práce.

2.2.1 Projekce

Projekce je unární operátor relační algebry a zapisuje se $\Pi_{a_1,...,a_n}(R)$, kde:

- $a_1, ..., a_n$ označuje seznam atributů,
- R je vstupní tabulka.

Výsledkem projekce je vstupní tabulka bez sloupců, které nejsou v seznamu atributů. V Tabulce 1 máme tabulku `Osoba` s atributy `Jméno`, `Věk` a `Váha`. Tabulka 2 pak představuje výsledek projekce atributů `Věk` a `Váha`.

Stojí za povšimnutí, že v Tabulce č. 2 máme 2x n-tici `34, 80`. Toto by se v tradiční relační algebře nemohlo stát, protože ta duplicity nedovoluje.

$Osoba$		
Jmeno	Vek	Vaha
Pepa	34	80
Sára	28	64
Jiří	29	70
Lenka	54	54
Petr	34	80

Tabulka 1: Tabulka Osoba

$\Pi_{Vek,Vaha}(Osoba)$	
Vek	Vaha
34	80
28	64
29	70
54	54
34	80

Tabulka 2: Projekce atributů věk a váha nad tabulkou Osoba

Význam projekce v této práci však nespočívá pouze ve výběru určitých atributů. U projekce nemusí být uveden seznam atributů, ale obecněji seznam výrazů. Můžeme tak určitý způsobem manipulovat s hodnotou atributu.

$\Pi_{Vek+10,Vaha/2}(Osoba)$	
Vek	Vaha
44	40
38	32
39	35
64	27
44	40

Tabulka 3: Projekce atributů věk+10 a váha/2 nad tabulkou Osoba

V Tabulce č. 3 jde vidět jak můžeme pomocí projekce manipulovat s daty. Tato funkcionalita však není implementovaná.

2.2.2 Selekce

Selekce je v relační algebře unární operátor zapsaný jako $\sigma_{a\theta b}(R)$ nebo $\sigma_{a\theta v}(R)$, kde

- a a b jsou názvy atributů,
- θ je booleovský výraz,
- v je konstanta,
- R je zdrojová tabulka.

Často se můžeme setkat s označením, kdy θ zastupuje jednu z binárních operací porovnání. Pro naše účely jsme tuto definici upravili, protože obecně jazyky jako SQL každý booleovský berou jako podmínku selekce spíše než její omezení na jednoduché porovnání.

$Osoba$

Jmeno	Vek	Vaha
Pepa	34	80
Sára	28	64
Jiří	29	70
Lenka	54	54
Petr	34	80

Tabulka 4: Tabulka `Osoba`

$\sigma_{Vek \geq 34}(Osoba)$

Jmeno	Vek	Vaha
Pepa	34	80
Lenka	54	54
Petr	34	80

Tabulka 5: Selekce nad tabulkou `Osoba` podle atributu `Vek`

$\sigma_{Jmeno='Lenka' \vee Vaha=64}(Osoba)$

Jmeno	Vek	Vaha
Lenka	54	54
Sára	28	64

Tabulka 6: Selekce podle složené podmínky.

V Tabulce č. 5 jsou data, kde je hodnota sloupce `Vek` větší nebo rovna hodnotě 34. V Tabulce č. 6 vidíme selekci dat, kde jméno v Tabulce č. 4 je Lenka nebo váha osoby je 64.

2.2.3 Sjednocení

Sjednocení je binární operátor a zapisuje se jako $A \bigcup B$. Výsledkem je množina (resp. tabulka), která obsahuje všechny záznamy z obou tabulek A a B. Pro relační algebru platí, že množiny (tabulky) musí být *kompatibilní* – tzn. obě musí obsahovat stejné atributy.

$OsobaA$

Jmeno	Vek	Vaha
Pepa	34	80
Sára	28	64
Jiří	29	70

Tabulka 7: Osoba A

$OsobaB$

Jmeno	Vek	Vaha
Alfred	70	97
Jiří	29	70
Xavier	80	60
Morgana	342	55

Tabulka 8: Osoba B

$A \bigcup B$

Jmeno	Vek	Vaha
Pepa	34	80
Sára	28	64
Jiří	29	70
Alfred	70	97
Xavier	80	60
Morgana	342	55

Tabulka 9: Sjednocení tabulek Osoba A a Osoba B

Ve výsledku (Tabulka č. 9) vidíme sloupce z Tabulky č. 8 připojené za data z Tabulky č. 7. V jazyku SQL odpovídá sjednocení příkaz `UNION` a námi definované sjednocení v SQL odpovídá příkazu `Union all`.

2.2.4 Průnik

Průnik je také binární operátor a zapisuje se jako $A \bigcap B$. Výsledkem je množina (resp. tabulka), která obsahuje pouze prvky obsažené v obou těchto množinách. Při

$OsobaA$		
Jmeno	Vek	Vaha
Pepa	34	80
Sára	28	64
Jiří	29	70

Tabulka 10: Osoba A

$OsobaB$		
Jmeno	Vek	Vaha
Alfred	70	97
Pepa	34	80
Sára	28	64

Tabulka 11: Osoba B

$A \bigcap B$		
Jmeno	Vek	Vaha
Pepa	34	80
Sára	28	64

Tabulka 12: Průnik tabulek Osoba A a Osoba B

V Tabulce č. 12 vidíme výsledek průniku. Vidíme zde pouze řádky, které jsou zároveň v obou tabulkách 10 a 11).

2.2.5 Množinový rozdíl

Množinový rozdíl zapisujeme jako $B \backslash A$. Někdy je také nazýván jako relativní doplněk. Rozdíl, stejně jako sjednocení, musí splňovat zmíněnou podmínku kompatibility (viz 2.2.3). Rozdíl obsahuje všechny záznamy, které se vyskytují v relaci B, kromě záznamů, které se zároveň vyskytují v relaci A.

$OsobaA$		
Jmeno	Vek	Vaha
Pepa	34	80
Sára	28	64
Jiří	29	70

Tabulka 13: Osoba A

$OsobaB$		
Jmeno	Vek	Vaha
Alfred	70	97
Pepa	34	80
Sára	28	64

Tabulka 14: Osoba B

$A \backslash B$		
Jmeno	Vek	Vaha
Jiří	29	70
Alfred	70	97

Tabulka 15: Rozdíl tabulek Osoba A a Osoba B

Ve výsledné Tabulce č. 15 vidíme pouze řádky z Tabulek č. 13 a 14, které se nachází pouze v jedné z nich.

2.2.6 Kartézský součin

Kartézský součin je binární množinový operátor který se zapisuje jako $A \times B$. Výsledkem je množina obsahující všechny kombinace řádku z obou vstupních množin. Vstupní množiny nemusí mít žádný společný atribut.

Zamestnanec

Jmeno	ID	PracujeVOddeleni
Pepa	13	Nákup
Morgana	42	Prodej

Tabulka 16: Zaměstnanec

Ovoce

PorCislo	Nazev	Puvod
1	Rajce	Evropa
2	Ananas	Afrika
3	Kiwi	Novy Zeland

Tabulka 17: Ovoce

Zamestnanec $\times$ *Ovoce*

Jmeno	ID	Oddeleni	PorCislo	Nazev	Puvod
Pepa	13	Nákup	1	Rajce	Evropa
Pepa	13	Nákup	2	Ananas	Afrika
Pepa	13	Nákup	3	Kiwi	Novy Zeland
Morgana	42	Prodej	1	Rajce	Evropa
Morgana	42	Prodej	2	Ananas	Afrika
Morgana	42	Prodej	3	Kiwi	Novy Zeland

Tabulka 18: Příklad kartézského součinu mezi Tabulkami `Zaměstnanec` a `Ovoce`

Zde vidíme, že v tabulce č. 18 jsou spojeny záznamy z Tabulek č. 16 a 17 i přes to, že spolu nedávají smysl. Kartézský součin je využit u operátoru Cross join a stejně jako ostatní operátory tato implementace povoluje duplicitní záznamy, protože pracujeme s tabulkami.

2.2.7 Přejmenování

Mezi další unární operátory relační algebry patří přejmenování. Zapisuje se $\rho_{a/b}(R)$, kde a je nový název atributu, b je původní název a R je tabulka, nad kterou přejmenování provádíme. Výsledná tabulka po přejmenování bude obsahovat stejné záznamy jako tabulka původní, dojde pouze k přejmenování atributu a na atribut b.

$Osoba$

Jmeno	Vek	Vaha
Pepa	34	80
Sára	28	64
Jiří	29	70

Tabulka 19: Osoba

$\rho_{Jmeno/Name}(Osoba)$

Name	Vek	Vaha
Pepa	34	80
Sára	28	64
Jiří	29	70

Tabulka 20: Přejmenování sloupce Jméno v tabulce Osoba

V Tabulce č. 20 vidíme změnu v názvu prvního sloupce z Tabulky 19.

2.2.8 Spojení (Join)

Spojení existuje více typů. Nejčastěji se používá tzv. *přirozené spojení*. Dále existuje *theta-spojení* a tzv. *equijoin*, což je speciální případ theta-spojení. V naší práci pod spojením rozumíme právě equijoin, který zapisujeme jako $R \bowtie_{a=b} S$ kde

- a a b jsou názvy atributů,
- R a S jsou zdrojové tabulky.

Pro příklad můžeme použít tabulky `Zaměstnanec` a `Oddělení`.

$Zamestnanec$

Jmeno	ID	PracujeVOddeleni
Pepa	13	Nákup
Frank	69	Finance
Morgana	42	Prodej
Sára	84	Finance

Tabulka 21: Zaměstnanec

$Oddeleni$

ID	Oddeleni	Manazer
1	Prodej	Franta
2	Finance	Jiří
3	Nákup	Kateřina

Tabulka 22: Oddělení

$Zamestnanec \bowtie_{PracujeVOddeleni=Oddeleni} Oddeleni$

Jmeno	ID	PracujeVOddeleni	ID_2	Oddeleni	Manazer
Pepa	13	Nákup	3	Nákup	Kateřina
Frank	69	Finance	2	Finance	Jiří
Morgana	42	Prodej	1	Prodej	Franta
Sára	84	Finance	2	Finance	Jiří

Tabulka 23: Příklad spojení Tabulek Zaměstnanec a Oddělení

V tabulkách výše je spojení provedeno mezi sloupci *PracujeVOddeleni* a *Oddeleni* z tabulek 21 a 22). Díky tomu je výsledek kombinací zaměstnanců a oddělení (Tabulka 23). Také jde vidět, že po spojení tabulek se stejnými názvy se k jednomu sloupci přidá postfix `_2`.

Spojení lze rozložit pomocí operátorů kartézského součinu a selekce. Kartézský součin může produkovat relativně velký mezivýsledek, který je následně ihned redukován selekcí. Spojení lze efektivně řešit jako jednu operaci.

2.2.9 Vnější spojení

Vnější spojení je typ spojení, které se zapisuje jako $R \bowtie_{a=b \vee b=\emptyset} S$ kde

- a a b jsou názvy atributů,
- $\emptyset$ je prázdná hodnota,
- R a S jsou zdrojové tabulky.

Existují tři typy vnějšího spojení - *levé*, *pravé* a *úplné*. Levé a pravé spojení určuje, která tabulka zůstane celá a která se k ní připojí. Úplné spojení ponechá obě tabulky celé, ale spojí ekvivalentní záznamy. Úplné vnější spojení není v našem procesoru implementované.

Jako příklad můžeme použít tabulky `Zaměstnanec` (24) a `Oddělení` (25).

Zamestnanec

Jmeno	ID	PracujeVOddeleni
Pepa	13	Nákup
Frank	69	Finance
Morgana	42	Prodej
Sára	84	HR

Tabulka 24: Zaměstnanec

Oddeleni

ID	Oddeleni	Manazer
1	Prodej	Franta
2	Finance	Jiří
3	Nákup	Kateřina

Tabulka 25: Oddělení

$Zamestnanec \bowtie_{PracujeVOddeleni=Oddeleni \vee Oddeleni=\emptyset} Oddeleni$

Jmeno	ID	PracujeVOddeleni	ID_2	Oddeleni	Manazer
Pepa	13	Nákup	3	Nákup	Kateřina
Frank	69	Finance	2	Finance	Jiří
Morgana	42	Prodej	1	Prodej	Franta
Sára	84	HR			

Tabulka 26: Příklad vnějšího spojení tabulek `Zaměstnanec` a `Oddělení`

V tabulkách výše je spojení provedeno mezi sloupci *PracujeVOddeleni* a *Oddeleni*. Jelikož je využito (v tomto případě levé) vnitřní spojení, je poslední řádek v Tabulce (26) naplněný pouze daty z levé tabulky, protože v pravé tabulce k nim neexistuje ekvivalent.

2.2.10 Setřídění

Setřídění je operátor, který byl vytvořen pro potřeby implementace. V relační algebře by neměl význam, protože pracuje s relacemi, kde nemá smysl uvažovat o pořadí. Provizorně jej budeme značit jako †.

$Zamestnanec$

Jmeno	ID	PracujeVOddeleni
Pepa	13	Nákup
Frank	69	Finance
Morgana	42	Prodej
Sára	84	Finance

Tabulka 27: Zaměstnanec

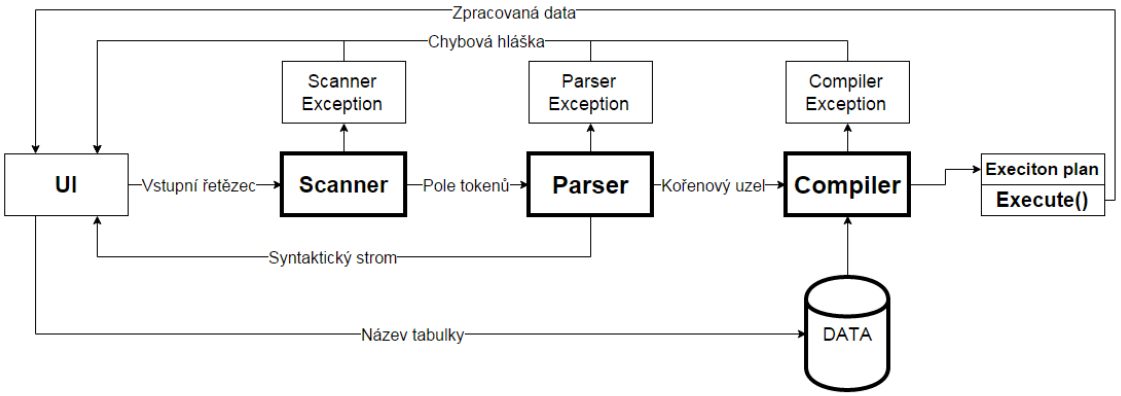
$\dagger_{Jmeno} Zamestnanec$

Jmeno	ID	PracujeVOddeleni
Frank	69	Finance
Morgana	42	Prodej
Pepa	13	Nákup
Sára	84	Finance

Tabulka 28: Oddělení

V Tabulce č. 28 vidíme data seřazená podle sloupce Jméno z Tabulky č. 27

3 Návrh architektury procesoru



Obrázek 2: Blokové schéma procesoru.

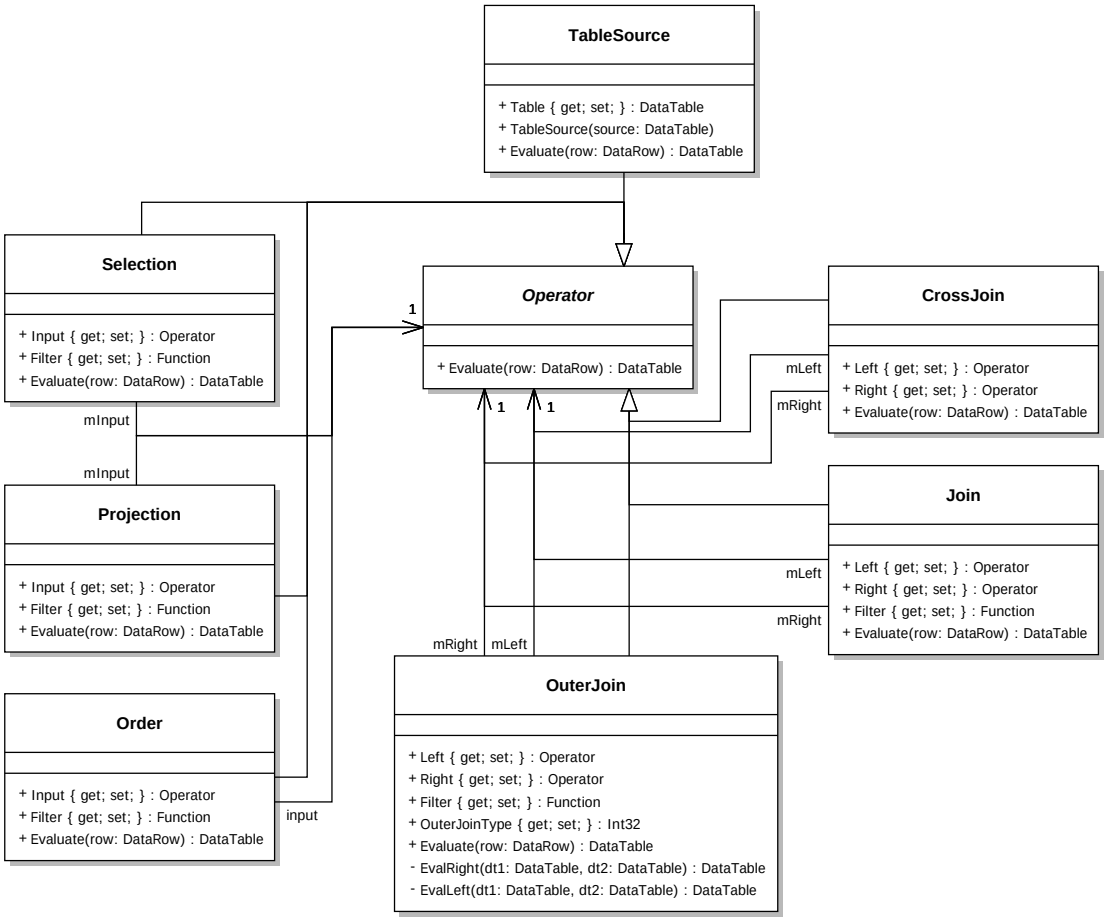
Zpracování dotazu začíná zadáním dotazu uživatelem pomocí grafického rozhraní. Zadaný dotaz je v podobě textového řetězce předán bloku Scanner (6.1.1), který provádí lexikální analýzu, tzn. rozděluje dotaz na posloupnost lexikálních jednotek (tzv. *tokenů*). Pole tokenů je následně předáno bloku Parser (6.1.2), který vytvoří operátorový strom. Výsledný strom zobrazí v grafickém rozhraní a kořenový uzel předá bloku Compiler (6.1.3) který se stará o vytvoření plánu dotazu. *Plán dotazu* je seřazená množina kroků, použitá k přistupování k datům. Kořenový operátor plánu je následně vyhodnocen, což spočívá v rekurzivním volání vyhodnocení podoperátorů. Tyto bloky jsou reprezentovány třídami `Scanner`, `Parser` a `Compiler`. Při zpracování dotazu jednotlivými bloky může dojít k chybě – může jít o chybu na úrovni Scanneru (neplatná posloupnost lexikálních symbolů), Parseru (neplatná posloupnost tokenů) nebo Compileru (chyba ve zpracování operátorového stromu). V případě chyby je generována výjimka, kterou zachytí uživatelské rozhraní.

3.1 Operátory a funkce

V předchozí kapitole byl definován pojem plán dotazu. Plán dotazu je v programu tvořen instancemi tříd, které dědí ze třídy `Operator` (viz Obrázek č. 3). Operátory mají jako vstupy jiné operátory, čímž vzniká stromová struktura.

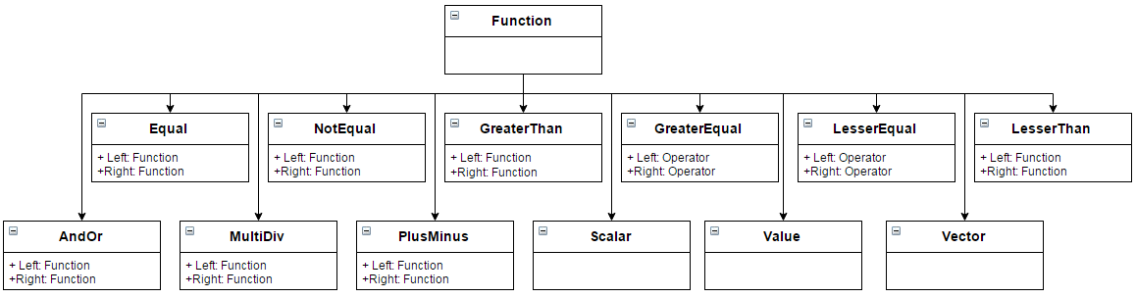
Některé operátory mají vstupní vlastnost *funkce*. Operátory využívají funkce k vyhodnocování booleovských výrazů a dále pro práci s názvy sloupců, případně hodnotou

v řádku. Rozdíl mezi operátorem a funkcí je v rozdílné práci s daty. Funkce data vrací nebo porovnává, kdežto operátor mění strukturu strukturu.



Obrázek 3: Třídní diagram operátorů

Třídy operátorů dědí z abstraktní třídy *Operator* metodu `Evaluate()` (viz 3). Každý operátor má rozšiřující vlastnosti a funkce podle potřeby daného operátoru. Stejným způsobem fungují i třídy funkcí.



Obrázek 4: Třídní diagram funkcí

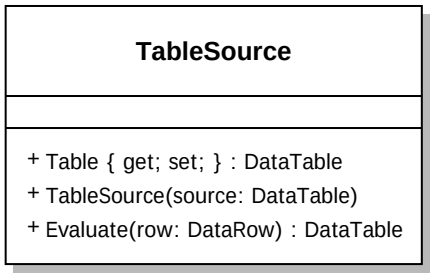
3.2 Použitý programovací jazyk a struktury

Pro implementaci byla zvolena platforma .NET, konkrétně jazyk C#, který je v současnosti velmi rozšířený. Jako datový model jsou v procesoru použity třídy .NET frameworku `DataTable`, `DataView`, `DataColumn` a `DataRow`. Třída `DataTable` je prvek knihovny ADO.NET a reprezentuje jednu tabulku v paměti. Objekt `DataView` slouží k vytvoření editovatelného pohledu nad určitou tabulkou `DataTable`. Především se využívá k seřazení, hledání a navigaci v této tabulce. Objekt `DataColumn` reprezentuje schéma sloupců tabulky a společně s `DataRow`, reprezentující řádky, se jedná o hlavní prvky objektu `DataTable`.

4 Implementace operátorů

Dle diagramu na Obrázku č.3, procesor pracuje se třídami, které dědí z abstraktní třídy `Operator`. Tato třída má tři metody. Nejpodstatnější je metoda `Evaluate()`, což je abstraktní metoda, kterou má každý operátor implementovanou podle jeho účelu. Dále metoda `CopyColumns(DataTable input)`, která kopíruje sloupce ze zadané tabulky typu DataTable a vrací je jako novou tabulku. Na rozdíl od metody `DataTable.CopyTo()` tato metoda nekopíruje data, pouze strukturu tabulky. Poslední metoda `AllColumns()` s parametry `input, output` oba datového typu `DataTable` přidá sloupce z jedné tabulky do druhé. Metoda je využita u operátorů spojení. Metody `CopyColumns()` a `AllColumns()` automaticky přejmenují duplicitní sloupce přidáním prefixu `_2`, což je výhodné u operátoru *Join*, pokud se názvy sloupců shodují.

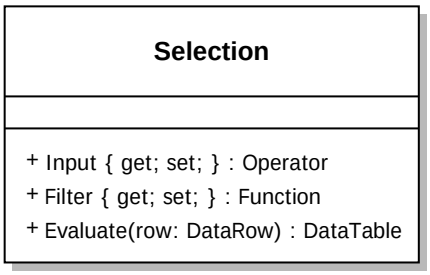
4.1 TableSource



Obrázek 5: Diagram třídy `TableSource`

Účelem tohoto operátoru je pouze vracet tabulku nastavenou ve vlastnosti `Table`. V relační algebře můžeme jako argumenty operátorů používat přímo názvy relací. Z hlediska implementace však každý argument operátoru musí být opět operátor (plán dotazu (viz kapitola...) je tvořen pouze operátory).

4.2 Selekce



Obrázek 6: Diagram třídy `Selection`

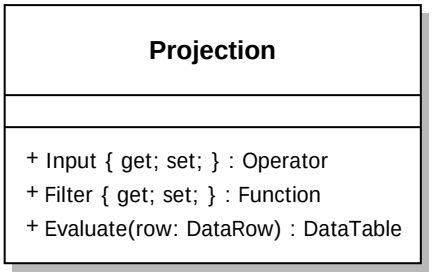
Operátor `select` má za úkol vyhodnotit booleovskou podmínku a podle ní vybrat řádky z tabulky. (viz 2.2.2)

Algoritmus 1 Metoda `Evaluate()` třídy `Selection`

1: $input = Input.Evaluate(row)$
2: $output = CopyColumns(input)$
3: **for all** row in $input.Rows$ **do**
4: **if** $filter.Evaluate() = true$ **then**
5: importuje row do $output$
6: **end if**
7: **end for**
8: **return** $output$

V algoritmu č. 1 vidíme popsanou mertodu `Evaluate()`. Nejprve vyhodnotíme vstupní operátor (řádek 1) čímž získáme tabulku `input`. Z té následně zkopírujeme sloupce do tabulky `output` (řádek 2) pomocí metody `CopyColumn(DataTable dt)`. Na řádcích 3–7 procházíme všechny řádky tabulky `input`. Dále se zkontroluje, zda je vyhodnocení vlastnosti `Filter` rovno `true` (řádek 4). V případě, že ano, importuje se řádek `row` do tabulky `output` (řádek 5). Nakonec vrátíme výslednou tabulku (řádek 8).

4.3 Projekce



Obrázek 7: Diagram třídy `Projection`

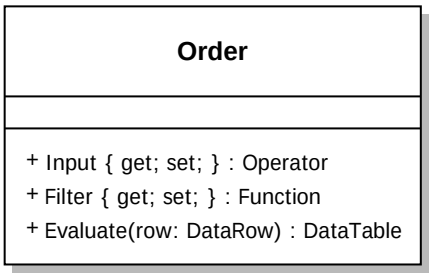
Operátor projekce má za úkol zobrazit tabulku pouze s uvedenými sloupci. (viz 2.2.1)

Algoritmus 2 Metoda `Evaluate()` třídy `Projection`

```
1:  input = Input.Evaluate(row)
2:  arr = Filter.Evaluate(row, null)
3:  output = input.Copy()
4:  doWant = false
5:  for all column v input.Columns do
6:      for all colName v arr do
7:          if column.Name je rovno colname then
8:              doWant = true
9:          end if
10:     end for
11:     if doWant = false then
12:         smaže column z output
13:     end if
14:     doWant = false
15: end for
16: return output
```

V algoritmu č. 2 nejprve vyhodnotíme vstupní operátor (řádek 1), čímž získáme tabulku `input`. Tu následně zkopírujeme do tabulky `output` (řádek 3). Na řádcích 5–15 procházíme všechny sloupce tabulky `input`. Jestliže sloupec není mezi atributy v projekci (řádky 6–10), odstraníme sloupec z tabulky `output` (řádky 11–13). Nakonec vrátíme výslednou tabulku (řádek 16).

4.4 Order



Obrázek 8: Diagram třídy `Order`

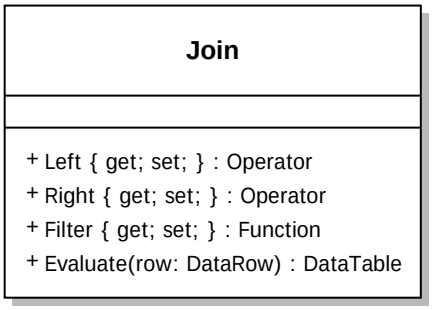
Operátor `Order` má za úkol setřídit tabulku podle zadaných sloupců tabulky (viz 2.2.10).

Algoritmus 3 Metoda `Evaluate()` třídy `Order`

1: $dt = Input.Evaluate(row)$
2: $arr = Filter.Evaluate(row, null)$
3: **for all** $colName$ v arr **do**
4: přidá $colName$ + $'$,$'$ do $inputStr$
5: **end for**
6: vytvoří DataView dw nad tabulkou dt.
7: setřídí dw podle $inputStr$
8: převede dw zpět do tabulky dt
9: **return** dt

V metodě `Evaluate()`, popsané Algoritmem č. 3 nejprve vyhodnotíme vstupní operátor (řádek 1), čímž získáme tabulku `dt`. Následně zjistíme seznam atributů, podle kterých budeme provádět setřídění (řádek 2). Na řádcích 3–5 procházíme jednotlivé atributy uložené v poli `arr` a řetězíme je do `inputStr`, který bude sloužit jako parametr pro seřazení. Dále vytvoříme objekt `DataView dw` nad tabulkou `dt` (řádek 6), který seřadíme podle zmíněného řetězce (řádek 7) a výsledek převedeme zpět do tabulky dt (řádek 8). Nakonec vracíme výslednou tabulku (řádek 9). Samotný algoritmus setřídění řádků jsem neimplementoval, ale využil jsem metodu `Sort` ve třídě `DataView`, která využívá třídící metodu *QuickSort*.

4.5 Join – spojení



Obrázek 9: Diagram třídy `Join`

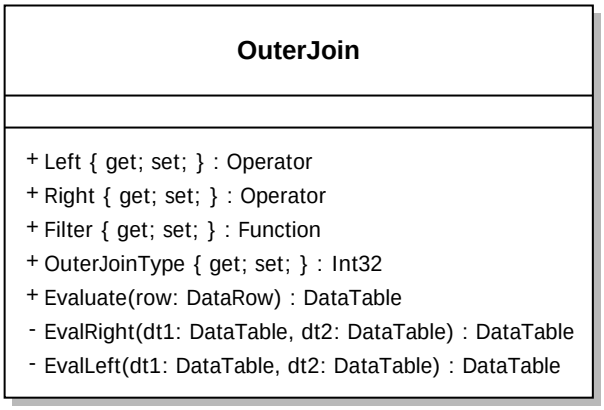
Operátor `Join` má za úkol spojit dvě zadané tabulky na základě rovnosti dvou zadaných sloupců nacházejících se v těchto tabulkách. Jak již bylo uvedeno, jedná se o variantu spojené na základě rovnosti atributů, tedy *equijoin* (viz 2.2.8). Pro spojování tabulek existuje několik *spojovacích algoritmů*. Mezi nejznámější patří *hash join*, *sort-merge join* a *nested loop join*, který jsme využili při implementaci.

V metodě `Evaluate()`, popsané Algoritmem č. 4 nejprve vyhodnotíme vstupní operátory (řádek 1–2), čímž získáme tabulky `dt1` a `dt2`. Dále do tabulky `output` zkopírujeme sloupce z `dt1` (řádek 3) a přidáme sloupce z `dt2` (řádek 4). Na řádcích 5–15 procházíme všechny řádky tabulky `dt1`. Jestliže je hodnota sloupce v aktuálním řádku `r1` z `dt1` rovna hodnotě sloupce aktuálního řádku `r2` z `dt2` (řádek 8), vložíme postupně oba řádky do nového řádku `newRow` (řádky 9–11). Následně vložíme `newRow` do tabulky `output` (řádek 12) a tabulku `output` vrátíme (řádek 16).

Algoritmus 4 Metoda `Evaluate()` třídy `Join`

1: $dt1 = Left.Evaluate(row)$
2: $dt2 = Right.Evaluate(row)$
3: zkopíruje sloupce z $dt1$ do $output$
4: do $output$ přidá sloupce $dt2$
5: **for all** row v $dt1$ **do**
6: vytvoří nový DataRow $newRow$
7: **for all** row in $dt2$ **do**
8: **if** $Filter.Evaluate(r1, r2)$ is $true$ **then**
9: **for** od $i = 0$ do $i < outputcolumncount$ **do**
10: nastaví všechny hodnoty do $newRow$
11: **end for**
12: přidá $newRow$ do $output$
13: **end if**
14: **end for**
15: vynuluje $newRow$
16: **end for**
17: **return** dt

4.6 Outer Join – Vnější spojení



Obrázek 10: Diagram třídy `OuterJoin`

Operátor *Outer Join* funguje podobně jako *Join*, s rozdílem že z jedné tabulky vezme všechny řádky a k nim přiřadí podle rovnosti hodnot v zadaných sloupcích řádky z druhé

tabulky. Která tabulka se má vzít celá, se pozná podle atributu `outerJoinType`. Hodnoty atributů z připojené tabulky pro řádky, kde neexistuje ekvivalent zůstanou prázdné (viz 2.2.9). Metoda `Evaluate()` třídy `OuterJoin` nejprve vyhodnotí obě vstupní tabulky (vlastnosti `Left` a `Right`), které následně předá metodě `EvaluateLeft` nebo `EvaluateRight` (viz 4.6.1 a 4.6.2).

4.6.1 Left Outer Join – Levé vnější spojení

Levá tabulka zůstává zachována a přiřazují se k ní řádky z tabulky pravé.

Algoritmus 5 Metoda `EvaluateLeft()`

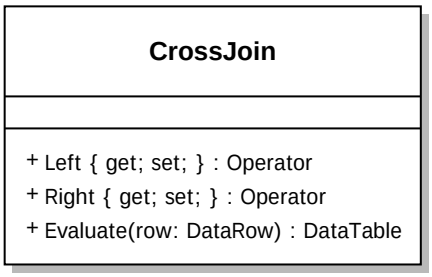
```
1:  zkopíruje dt1 do output
2:  přidá sloupce z dt2 do output
3:  for all rows v output do
4:      for all rows v dt2 do
5:          if Filter.Evaluate(rOut, r2) is true then
6:              for od i = 0 do i < dt2columncount do
7:                  nastaví všechny hodnoty do aktuálního řádku v output
8:              end for
9:          end if
10:     end for
11: end for
12: return output
```

Tato metoda, popsaná v Algoritmu č. 5 je volána metodou `Evaluate()`, od které dostane vytvořené tabulky `dt1` a `dt2`. Nejprve se tabulka `dt1` nakopíruje do tabulky `output` (řádek 1). Přidají se sloupce z tabulky `dt2` (řádek 2) a na řádcích 3–10 se prochází všechny řádky tabulky `output`. Pak probíhá porovnání, zda jsou hodnoty v zadaných sloupcích v obou řádcích shodné (řádek 5) a pokud ano, přidají se k aktuálnímu řádku hodnoty z tabulky `dt2` (řádky 6–8). Nakonec se vrátí tabulka `output` (řádek 12).

4.6.2 Right Outer Join – Pravé vnější spojení

Chování této metody je téměř shodné s metodou `EvaluateLeft()` s tím rozdílem, že celá zůstává tabulka pravá a přiřazují se k ní řádky z tabulky levé.

4.7 Cross Join



Obrázek 11: Diagram třídy `CrossJoin`

`Cross join` představuje kartézský součin řádků z obou tabulek a tím pádem není třeba řešit jestli se řádky vzájemně rovnají (viz. 2.2.6).

Algoritmus 6 Metoda `Evaluate()` třídy `CrossJoin`

1: $left = Left.Evaluate(row)$
2: $right = Right.Evaluate(row)$
3: zkopíruje sloupce z $left$ do $output$
4: přidá sloupce z $right$ do $output$
5: **for all** row v $left$ **do**
6: **for all** row v $right$ **do**
7: vytvoří nový DataRow $newRow$
8: **for** od $i = 0$ do $i < outputcolumncount$ **do**
9: nastaví všechny hodnoty do $newRow$
10: **end for**
11: přidá $newRow$ do $output$
12: **end for**
13: **end for**
14: **return** output

V Algoritmu č. 6 nejprve vyhodnotíme operátory (řádek 1–2), čímž získáme tabulky `left` a `right`. Dále nakopírujeme sloupce z tabulky `left` do tabulky `output` (čímž tuto tabulku inicializujeme) (řádek 3) a následně se do této tabulky přidají sloupce z tabulky `right` (řádek 4). Následně procházíme dvěma zanořenými cykly všechny řádky z tabulky `left` (řádky 5–13) a všechny řádky z tabulky `right` (řádky 6–12). Během každého průchodu vnořeného cyklu vytvoříme nový řádek `newRow` (řádek 7) a v cyklu

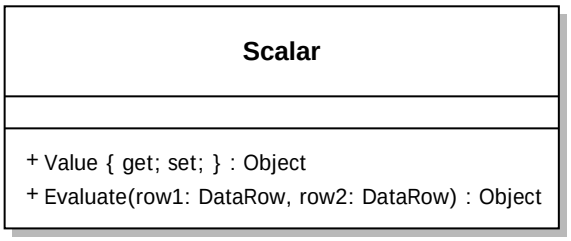
(řádek 8–10) nastavíme všechny hodnoty do `newRow` (řádek 9). Nakonec vrátíme tabulku `output` (řádek 14).

4.8 Funkce

V této kapitole popíšeme implementaci jednotlivých funkcí. Funkce slouží k vyhodnocování aritmetických a booleovských výrazů, které se používají např. v podmínkách selekcí, v atributech u projekce atd.

Z Obrázku 4 jde vidět, že funkce dědí z abstraktní třídy *Function*. Tato třída má vlastnosti *Name* a *ResultType*, které si každá dědící funkce nastaví v konstruktoru. Dále má metodu `Evaluate()`, kterou si každá funkce přetíží podle potřeby. Tato metoda má vstupní parametr – pole typu `DataRow`. Každý řádek má odkaz na tabulku, ze které pochází. V případě, že metoda `Evaluate()` kontroluje tabulku nebo nějaký její prvek, jedná se o tuto tabulku předaných řádků `row1` nebo `row2`.

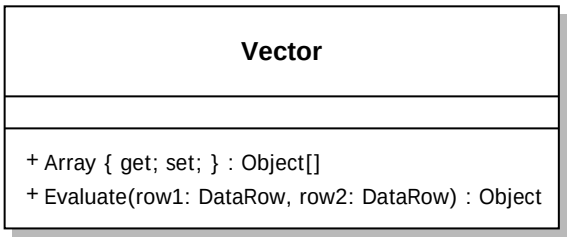
4.8.1 Scalar



Obrázek 12: Diagram třídy `Scalar`

Funkce `Scalar` je základní funkce, která má vlastnost `Value`, která se také nastavuje v konstruktoru třídy a metoda `Evaluate()` pouze vrací hodnotu nastavenou ve vlastnosti `Value`.

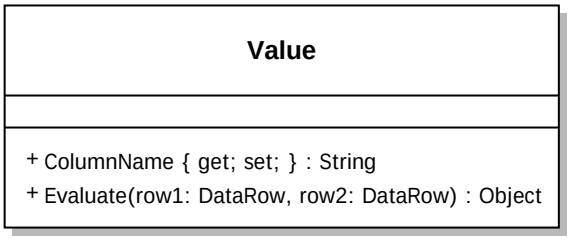
4.8.2 Vector



Obrázek 13: Diagram třídy `Vector`

Funkce *Vector* je pouze rozšířená funkce *Scalar*. Jediný rozdíl je, že se nejedná o samostatnou hodnotu, ale o pole hodnot uložené ve vlastnosti *Array*. Metoda `Evaluate()` opět pouze vrací pole uložené v *Array*.Tuto funkci využívají operátory `Project` a `Order` k uložení názvů sloupců, podle kterých zpracovávají data.

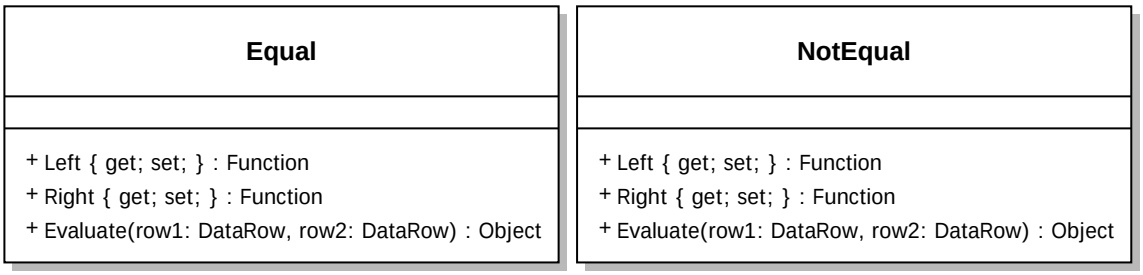
4.8.3 Value



Obrázek 14: Diagram třídy `Value`

Tato funkce má vlastnost *ColumnName* a hodnota se do ní ukládá v konstruktoru. Metoda `Evaluate()` zjistí, zda tabulka obsahuje sloupec se zadaným názvem a případně vrátí jeho hodnotu. V opačném případě je zachycena výjimka a uživateli je zobrazena hláška o absenci zadaného sloupce. Tuto funkci se využívají ostatní funkce, které porovnávají hodnoty v řádcích jako například `Equal`, `GreaterThan` a další.

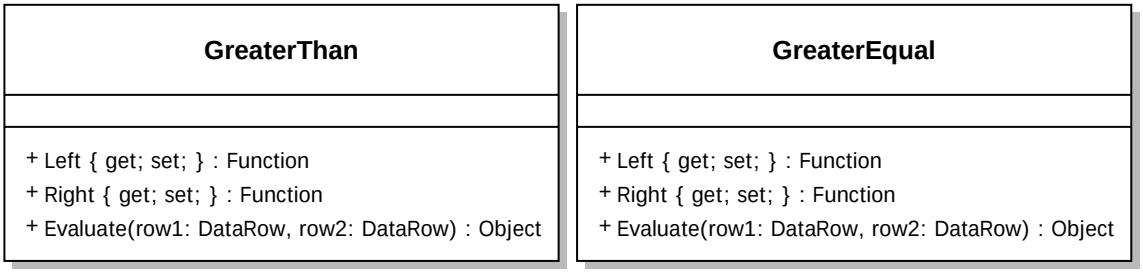
4.8.4 Rovná se a Nerovná se



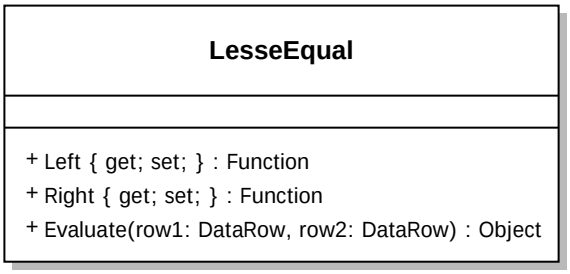
Obrázek 15: Diagram třídy `Equal` Obrázek 16: Diagram třídy `NotEqual`

Obě funkce mají dvě vlastnosti typu `Function` – `Left` a `Right` a u obou funkcí metoda `Evaluate()` provede evaluaci zmíněných vlastností a následně porovná jejich výsledky. Porovnání probíhá zavoláním metody `Equals` implementované v metodě `object`, které předáváme jako parametry výsledky evaluací vlastností `Left` a `Right`. Rozdíl nastává při vrácení výsledné hodnoty. Funkce `Equal` po splnění podmínky vrací `true`, kdežto `NotEqual false` a opačně. V případě, že by došlo k pokusu o porovnání dvou rozdílných datových typů bude zachycena výjimka a uživateli se zobrazí zpráva a nekompatibilitě typů.

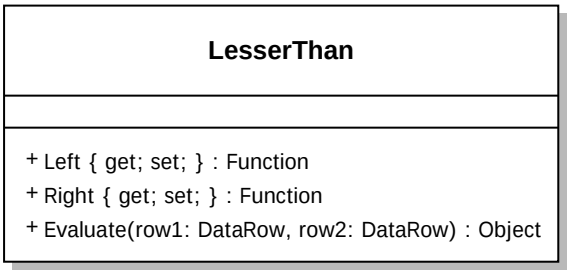
4.8.5 Větší/nebo rovno a Menší/nebo rovno



Obrázek 17: Diagram třídy `GreaterThan` Obrázek 18: Diagram třídy `GreaterEqual`



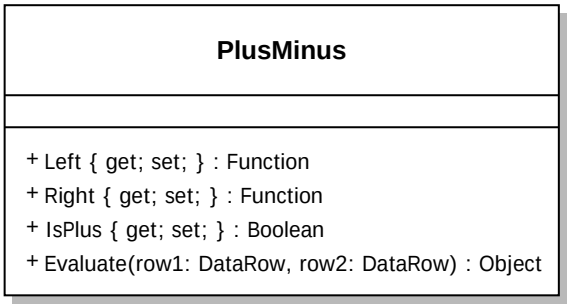
Obrázek 19: Diagram třídy `LessEqual`



Obrázek 20: Diagram třídy `LessThan`

Tyto funkce také mají vlastnosti *Left* a *Right* a metoda `Evaluate()` provede jejich evaluaci a dále porovná zda vzájemně splňují danou podmínku (<,<=,>,>=). Pokud je daná podmínka splněna, pak metoda vrátí `true`, v opačném případě `false`.

4.8.6 Sčítání a Odčítání



Obrázek 21: Diagram třídy `PlusMinus`

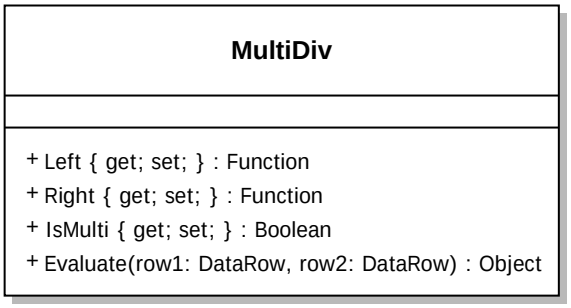
Tato funkce obsahuje vlastnosti `Left`, `Right` a vlastnost typu boolean `IsPlus`, která se nastavuje v konstruktoru.

V Algoritmu č. 7 nejprve proběhne evaluace vlastností `Left` a `Right` (řádek 1 a 2). Dále se provede kontrola, zda proměnná `IsPlus` je `true` (řádek 3). Pokud ano, tak se vrátí součet hodnot (řádek 4). V opačném případě se vrátí jejich odečet (řádek 6).

Algoritmus 7 Metoda `Evaluate()` třídy `PlusMinus`

1: $left = Left.Evaluate(row)$
2: $right = Right.Evaluate(row)$
3: **if** $isPlus$ **then**
4: **return** $left + right$
5: **else**
6: **return** $left - right$
7: **end if**

4.8.7 Násobení a Dělení



Obrázek 22: Diagram třídy `MultiDiv`

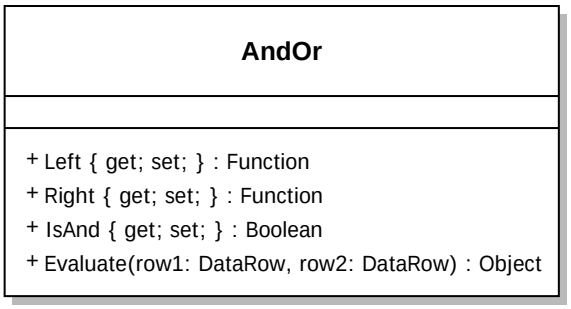
Tato funkce obsahuje vlastnosti `Left`, `Right` a vlastnost typu boolean `IsMulti`. Tato vlastnost se nastavuje v konstruktoru.

V prvním kroku Algoritmu č. 8 se provede evaluace vlastností `Left` a `Right` (řádek 1 a 2). Dále se provede kontrola, zda je proměnná `IsMulti` nastavená na `true` (řádek 3). Pokud ano, tak se vrátí součin hodnot (řádek 4). V opačném případě se zkontroluje, zda není pravá strana (dělitel) rovna nule a pokud ano, tak se vypíše chybová hláška. V opačném případě se vrátí podíl hodnot.

Algoritmus 8 Metoda `Evaluate()` třídy `MultiDiv`

1: $left = Left.Evaluate(row)$
2: $right = Right.Evaluate(row)$
3: **if** $isMulti$ **then**
4: **return** $left * right$
5: **else**
6: **if** $right \neq 0$ **then**
7: **return** $left/right$
8: **else**
9: Vygenerování vyjímky: Nelze dělit nulou.
10: **end if**
11: **end if**

4.8.8 And a Or



Obrázek 23: Diagram třídy `AndOr`

Tato funkce obsahuje vlastnosti `Left`, `Right` a vlastnost typu boolean `IsAnd`. Tato vlastnost se nastavuje v konstruktoru.

Algoritmus 9 Metoda `Evaluate()` třídy `AndOr`

```
1:  left = Left.Evaluate(row)
2:  right = Right.Evaluate(row)
3:  if IsAnd then
4:      if left ∧ right then
5:          return true
6:      else
7:          return false
8:      end if
9:  else
10:     if left ∨ right then
11:         return true
12:     else
13:         return false
14:     end if
15: end if
```

Metoda `Evaluate()` (viz Algoritmus č. 9) nejdříve provede evaluaci vlastností `Left` a `Right` (řádky 1 a 2), dále zkontroluje, zda je vlastnost `IsAnd` nastavena na `true` (řádek 3). Pokud ano, provede se kontrola,zda levá i pravá strana jsou zároveň `true` (řádek 4). V tomto případě metoda vrátí `true` (řádek 5), v opačném `false` (řádek 7). Pokud je `IsAnd` nastaveno na `false`, provede se porovnání, zda je alespoň jedna ze stran `true` (řádek 10). Pokud ano, vrátí hodnotu `true` (řádek 11), jinak vrátí hodnotu `false` (řádek 13).

5 Návrh dotazovacího jazyka využívajícího implementované operátory

Při návrhu jazyka bylo potřeba zvolit textové ekvivalenty k řeckým zápisům operátorů relační algebry. To by komplikovalo jak implementaci, tak zadávání dotazu uživatelem. Zavedená syntaxe se v mnoha ohledech blíží jazyku SQL.

Obrázek 24: Zápis operátoru

Příkaz vždy začíná klíčovým slovem operátoru. Podle konkrétního typu operátoru poté následuje seznam parametrů (výrazů). Příkaz končí specifikací vstupu(ů) operátoru. Syntaxe jednotlivých operátorů vypadají následovně:

Select	select *booleovský výraz vstup*
Project	project *seznam atributů vstup*
Order	order *seznam atributů vstup*
Join	join *booleovský výraz vstup1, vstup2*
OuterJoin	left outer join *booleovský výraz vstup1, vstup2*
nebo	right outer join *booleovský výraz vstup1, vstup2*
CrossJoin	cross join *vstup1, vstup2*

Tabulka 29: Tabulka definicí příkazů

Gramatika tohoto jazyka je inspirovaná gramatikou jazyka SQL. Implementovaný jazyk ignoruje malá a velká písmena. Kompletní gramatiku nalezneme na Obrázku č. 25.

QUERY → SELECT | PROJECT | LEFT | RIGHT | CROSS | JOIN | ORDER

SELECT → “select” “{“ EXPR “}” QUERY | TABLE

PROJECT → “project” “{“ FIELDS ”}” QUERY | TABLE

LEFT → “left“ OUTER

RIGHT → “right” OUTER

OUTER → “outer” JOIN

CROSS → “cross” JOIN

JOIN → “join” “{“ OREXPR “}” QUERY | TABLE

ORDER → “order” “{”FIELDS “}” QUERY | TABLE

TABLE → EXPRLIST

EXPR → OREXPR

OREXPR → ANDEXPR (“|” ANDEXPR)*

ANDEXPR → COMPEXPR (“&” COMPEXPR)*

COMPREXPR → ADDEXPR COMPFUNCT ADDEXPR

ADDEXRP → MULTIPEXPR (“+” | “-“ MULTIPEXPR)*

COMPFUNCT → “<” | “<=” | “=” | “!=” | “>=” | “>”

MULTIPEXPR → UNARYEXP (“*” | “/” UNARYEXP)*

UNARYEXPR → “-“ VALUEXPR

VALUEXPR → STRING | NUMBER | BOOL | IDENTIFIER | „(“ EXPR „)“

STRING → “%”*

NUMBER → (0-9)* “.” (0-9)*

BOOL → “true” | “false”

IDENTIFIER → STRING

EXPRLIST → STRING (“,” STRING)*

Obrázek 25: Gramatika implementovaného jazyka

6 Implementace jazyka a uživatelského rozhraní

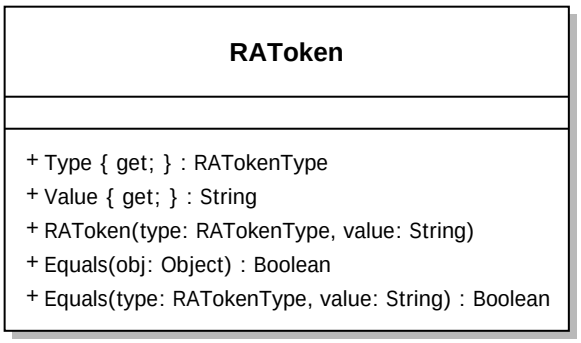
V této kapitole je popsána implementaci jazyka, ukážeme detailnější příklady použití programu a testování procesoru a následně porovnání našeho procesor s podobným programem.

6.1 Implementace jazyka

Popis implementace je rozdělen na tři části. První popisuje třídu *Scanner*, která zpracovává vstupní řetězec. Druhá fáze popisuje třídu *Parser*, která z výstupu třídy *Scanner* vytváří operátorový strom a nakonec máme popsanou třídu *Compiler*, která se stará o provedení požadovaného příkazu nad daty. Schéma najdeme na Obrázku č. 2.

6.1.1 Scanner

Na začátku bylo potřeba vyřešit zpracování vstupního řetězce. Tato část je vyřešena třídou `Scanner`, která se stará o procházení jednotlivých znaků ve vstupním řetězci a podle nich vytváří tzv. *Tokeny*. Pro tokeny máme třídu, kterou můžete vidět na Obrázku č. 26. Každý token má vlastní popisek a typ, díky kterému v programu dokážeme rozeznat, zda-li se jedná o klíčové slovo, identifikátor nebo různé typy hodnot, jako například čísla a řetězce. Toto řeší metoda `Scan()`.

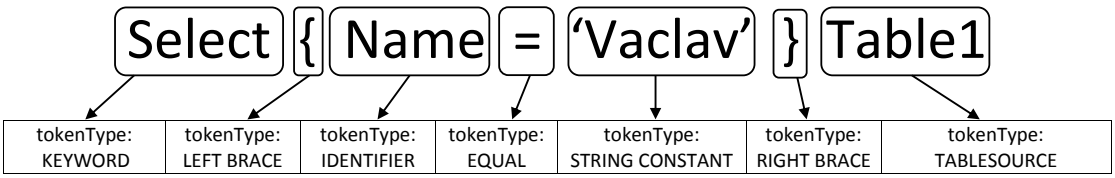


Obrázek 26: Diagram třídy `Token`

O procházení vstupního řetězce se stará metoda `Tokenize()`, která pro vytvoření tokenu volá metodu `CreateToken(string input)`. Nyní si tuto metodu popíšeme.

Nejprve se zkontroluje, zda je slovo jeden z dělících znaků (závorky, složené závorky, operátory, atd.), případně se vytvoří token s typem odpovídajícím danému dělícímu

znaku. Dále se ověří, jestli slovo začíná apostrofami. Pokud ano, jedná se o textový řetězec a vytvoří se token s typem `STRING_COSNTANT`. Slovo bez apostrof se zkontroluje, zda začíná číslicí. Pro takové slovo se vytvoří token s typem `NUMERIC_CONSTANT`. Pro slovo začínající písmenem se zavolá metoda `IsKeyword(slovo)`, která otestuje zda se jedná o klíčové slovo. Pokud je podmínka splněna, dojde k vytvoření tokenu s typem `KEYWORD`. V opačném případě se zkontroluje zda se jedná o zdroj v tabulce a pokud slovo neprojde ani touto kontrolou, vytvoří se token typu identifikátor. Následně je každý token přidán do seznamu a po dokončení smyčky procházející vstupní řetězec se celý list zakončí tokenem typu EOF.



Obrázek 27: Rozbor typů tokenů

6.1.2 Parser

Po ukončení metody `Scan()` třídy `Scanner` se pole tokenů předá metodě `Parse()`, která skládá jednotlivé tokeny do stromové struktury – *syntaktického stromu* podle navržené gramatiky. Základem této stromové struktury je abstraktní třída *Node*, která má proměnné pro uchovávání informace o svém předku, potomcích a dále vlastnost pro získání názvu uzlu, jeho typu a popisku. Z této třídy dále dědí třídy představující neterminály použité gramatiky.

Nejprve vždy dojde k nastavení tokenu do proměnné *currentToken* z předaného listu. Dále se volá metoda `ConsumeQuery()`, která podle typu aktuálního tokenu zavolá obsluhující metodu pro daný operátor nebo zachytí výjimku. Následná metoda vždy vytvoří první, kořenový, uzel a nastaví vlastnosti podle toho o jaký operátor se jedná. Pro demonstraci si popíšeme pseudokód metody `ConsumeJoin()` (viz Algoritmus č. 10). Proměnná `currentToken` je před zavoláním této metody nastavena na token `Join`.

Algoritmus 10 Metoda `ConsumeJoin()`

```
1:  vytvoření proměnné result typu RAOperator
2:  do proměnné prev nastaví předchozí token
3:  if currentToken je klíčové slovo then
4:      if hodnota prev = 'cross' then
5:          result = nová instanci třídy RACrossJoin
6:      else if hodnota prev = 'outer' then
7:          result = nová instanci třídy RAOuterJoin
8:      else
9:          result = nová instanci třídy RAJoin
10:     end if
11:     přesune do currentToken další token
12: else
13:     vypíše chybovou hlášku: nesprávný token
14: end if
15: if currentToken = { nebo hodnota prev = 'cross' then
16:     if currentToken = { then
17:         přesune do currentToken další token
18:         result.Expression = ConsumeExpression()
19:     end if
20: else
21:     vypíše chybovou hlášku: chybí levá závorka
22: end if
23: if currentToken = } nebo currentToken.Type je TABSRC then
24:     if currentToken = } then
25:         přesune do currentToken další token
26:     end if
27:     if currentToken.Type je TABSRC then
28:         result.Source = ConsumeTables()
29:     else
30:         vypíše chybovou hlášku: chybí tabulka
31:     end if
32: else
33:     vypíše chybovou hlášku: chybí pravá závorka
34: end if
35: return result
```

Nejprve se vytvoří nový uzel pro operátor (řádek 1). Dále se do proměnné `prev` nastaví předchozí token (řádek 2). Zkontroluje se, zda je `currentToken.Type` rovno typu `KEYWORD` (řádek 3). V případě, že ano, zkontroluje se, zda se hodnota proměnné `prev` rovná řetězci 'cross' (řádek 4) a pokud ano, do proměnné result se nastaví uzel pro operátor `CrossJoin` (řádek 5). Pokud hodnota `prev` není 'cross', provede se kontrola, zda je hodnota řetězec 'outer' (řádek 6). Pokud ano, do proměnné `result` se nastaví uzel pro operátor `OuterJoin` (řádek 7). V ostatních případech se do proměnné `result` nastaví uzel pro operátor `Join` (řádek 9). Pokud však typ aktuálního tokenu není `KEYWORD`, dojde k výpisu chybové hlášky (řádek 12), která uživateli oznámí, že je očekáváno klíčové slovo.

V druhé části proběhne kontrola, zda se `currentToken` rovná { nebo zda je předchozí token řetězec 'cross' (řádek 15). Pokud je podmínka splněna, zkontroluje se znovu zda se `currentToken` rovná { (řádek 16) a pokud ano, do proměnné currentToken se přesune následující token (řádek 17) a do vlastnosti `Expression` proměnné `result` se nastaví výsledek metody `ConsumeExpression()` (řádek 18). Pokud první podmínka není splněna, dojde k výpisu chybové hlášky, která oznámím, že chybí levá složená závorka (řádek 21).

V poslední části se zkontroluje, zda je `currentToken` = } nebo je typ proměnné `currentToken` roven typu `TABSRC` (řádek 23). Pokud je tato podmínka splněna, zkontroluje se zda se `currentToken` rovná } a pokud ano, do proměnné `currentToken` se vloží následující token (řádek 25). Dále se provede kontrola, zda je `currentToken.Type` typ `TABSRC` (řádek 27) a pokud ano, tak se do vlastnosti `Source` proměnné `result` nastaví výsledek metody `ConsumeTables()` (řádek 28). V opačném případě se vypíše chybová hláška, která uživateli oznámí, že chybí zdrojová tabulka. V případě, že není splněna první podmínka, dojde k vypsání chybové hlášky ohledně chybějící pravé složené závorky (řádek 33). Nakonec metoda vrátí proměnnou `result`.

6.1.3 Compiler

Třída *Compiler* má za úkol provést příkaz nad daty. `Compiler` jako první zavolá metodu `Compile()`, které se předá jako parametr kořenový uzel. Tato metoda má za úkol nastavit veškeré potřebné operátory a s nimi související funkce.

Algoritmus 11 Metoda `CompileJoin()`

```
1: nová instance join třídy Join
2: join.Filter = CompileFnc() s parametrem ChildNodes
3: do relSrc uloží potomka s názvem 'Relation Source'
4: if počet relSrc.ChildNodes není 2 then
5:     vypíše chybovou hlášku: join musí mít zadané dvě tabulky
6: end if
7: join.Left = Compile(relSrc.ChildNodes['Table_0'])
8: join.Right = Compile(relSrc.ChildNodes['Table_1'])
9: return join
```

V Algoritmus č. 11) vidíme popsanou metodu `ConsumeJoin()`. V prvním kroku se vytvoří nová instance třídy `Join` (řádek 1), v našem případě pojmenovaná `join`. Následuje nastavení vlastnosti `Filter` (řádek 2) metodou `CompileFnc()`. Tato metoda dostává jako parametr potomka s názvem "Expression", kde jsou uchovány informace o atributech. V dalším kroku se vytvoří proměnná `relSrc`, do které se uloží potomek s názvem "Relation Source"(řádek 3). V tomto uzlu jsou uloženy informace o tabulkách. Následuje kontrola, zda počet potomků v `relSrc` není roven dvěma (řádek 4). Pokud je tato podmínka splněna, vypíše se uživateli chybová hláška s informací, že operátor `Join` musí mít zadané dvě tabulky (řádek 5). V dalším kroku se operátoru `join` nastaví vlastnost `Left` zavoláním metody `Compile()` na potomka s názvem 'Table_0' (řádek 7). Stejný proces následně probíhá i pro vlastnost `Right` s potomkem 'Table_1' (řádek 8). Nakonec metoda vrátí operátor join (řádek 9).

6.1.4 Propojení částí

Spouštění a propojení všech částí probíhá ve třídě hlavního formuláře. Samotné zpracování se provede po zmáčknutí tlačítka Start v uživatelském rozhraní, kdy se zavolá metoda `ParseQuery()`. Tato metoda první vytvoří instanci třídy *Parser* se vstupním řetězcem jako parametr a zavolá jeho metodu `Parse()`. Po provedení této metody se provede aktualizace stromu s novým kořenovým uzlem. Následně se vytvoří instance třídy *Compiler*, nastaví se vlastnost *TableList* a vytvoří se instance třídy *Operator*, do které se nastaví výsledek metody `Compile()`, které se předá jako parametr kořenový uzel. Následně se provede nastavení komponenty *dataGridView* daty a to pomocí metody `Evaluate()` vytvořeného operátoru.

Také se zde provádí importování tabulek z CSV souboru do programu. Slouží k tomu vedlejší formulář, kde se nastaví název nové tabulky a zavolá se `openFileDialog`,

který otevírá CSV soubory. Poté zpět ve třídě hlavního formuláře proběhne přidání tabulky do programu.

7 Testování procesoru

Program byl testován na dvou přednastavených a jedné importované tabulce. Tab1 má formát 20 řádků × 3 sloupce, Tab2 15 řádků × 4 sloupce a dále jednu tabulku s 200 000 záznamy ve čtyřech sloupcích – každý pro jeden podporovaný datový typ. Testovací sestava, na které testy proběhly obsahuje osmi jádrový procesor AMD FX8320 na frekvenci 3.5Ghz. Operační paměť o velikosti 8GB a SSD disk připojený pomocí SATAIII. Operační systém běžící na zmiňované sestavě byl Microsoft Windows 7 Professional v 64–bitové verzi.

7.1 Test základních operátorů

V tabulce č. 30 vidíme test základních operátorů. Měření bylo provedeno 5x pro každý operátor. Nejhorší a nejlepší čas jsme eliminovali a ze zbylých tří hodnot jsme vypočítali průměrný čas.

Příkaz	Čas
`select { ID < 20 } Tab1`	0s 052ms
`project { ID, Name } Tab1`	0s 043ms
`order { Name } Tab1`	0s 064ms
`join { ID = ID } Tab1, Tab2`	0s 068ms
`left outer join { ID = ID } Tab1, Tab2`	0s 079ms
`right outer join { ID = ID } Tab1, Tab2`	0s 072ms
`cross join Tab1, Tab2`	00s 250ms

Tabulka 30: Test základních operátorů

7.2 Test kombinace operátorů

V následující tabulce 31 najdeme test tří různých operátorů v jednom příkazu na různě velkém počtu dat. Čas je opět vypočítaný průměr ze tří hodnot. V posledních třech řádcích najdete časy provádění ekvivalentního příkazu v `SQLite` [3] na stejnými daty, pouze v jazyce SQL.

order { Name } project { ID, Name } select { ID < X } test	
X	Čas
1000	0s 16ms
10 000	0s 92ms
50 000	4s 62ms
100 000	9s 32ms
200 000	18s 89ms
order { Name } project {ID,Name,Value,isValid} select { ID < X } test	
200 000	1m 03s 45ms
select * from SpeedTest where ID < X order by Name;	
1000	0s 579ms
50 000	27s 241ms
200 000	1m 52s 269ms

Tabulka 31: Test kombinací operátorů

Je očividné, že program SQLite vykonával stejné příkazy nad daty déle než námi vytvořený program. Test můžeme označit za úspěšný díky faktu, že i kdyby program SQLite běžel rychleji, pořád by mezi oběma procesory nebyl velmi velký rozdíl.

7.3 Test korektnosti překladače

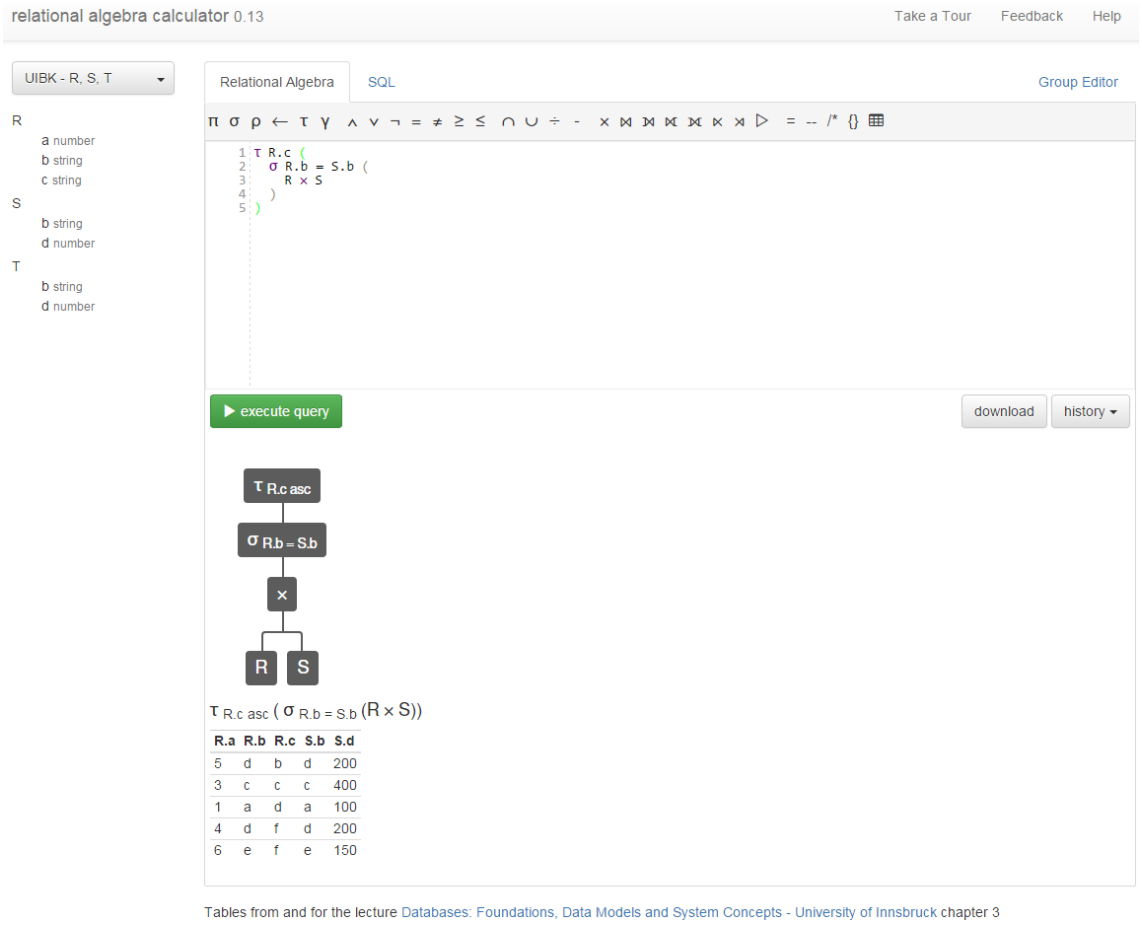
Následující výrazy nejsou dle gramatiky správně a překladač by na ně měl odpovídajícím způsobem reagovat.

- auglh Hláška: Operator expected.
- projec ID Tab1 Hláška: Operator expected
- project ID > 4 Tab1 Hláška: Parser: Unexpected token >
- select project ID Tab1 Hláška: Parser: Missing left brace.
- project ID, Name Tab12 Hláška: Succesfully processed but compile failed: table with name "Tab12"does not exist.
- select ID < 10 project Name Tab1 Hláška: Succesfully processed but compile failed: column ID doesn't exist.
- join id < id Tab1, Tab2 Hláška: Successfully processed but compile failed: Joins can only compare if columns are equal.

7.4 Porovnání s existujícími procesory

K porovnání jsme vybrali dva webové nástroje – Relational Algebra Calculator (dále jen RAC) [4] a ALF [5].

7.4.1 Relatinal algebra calculator



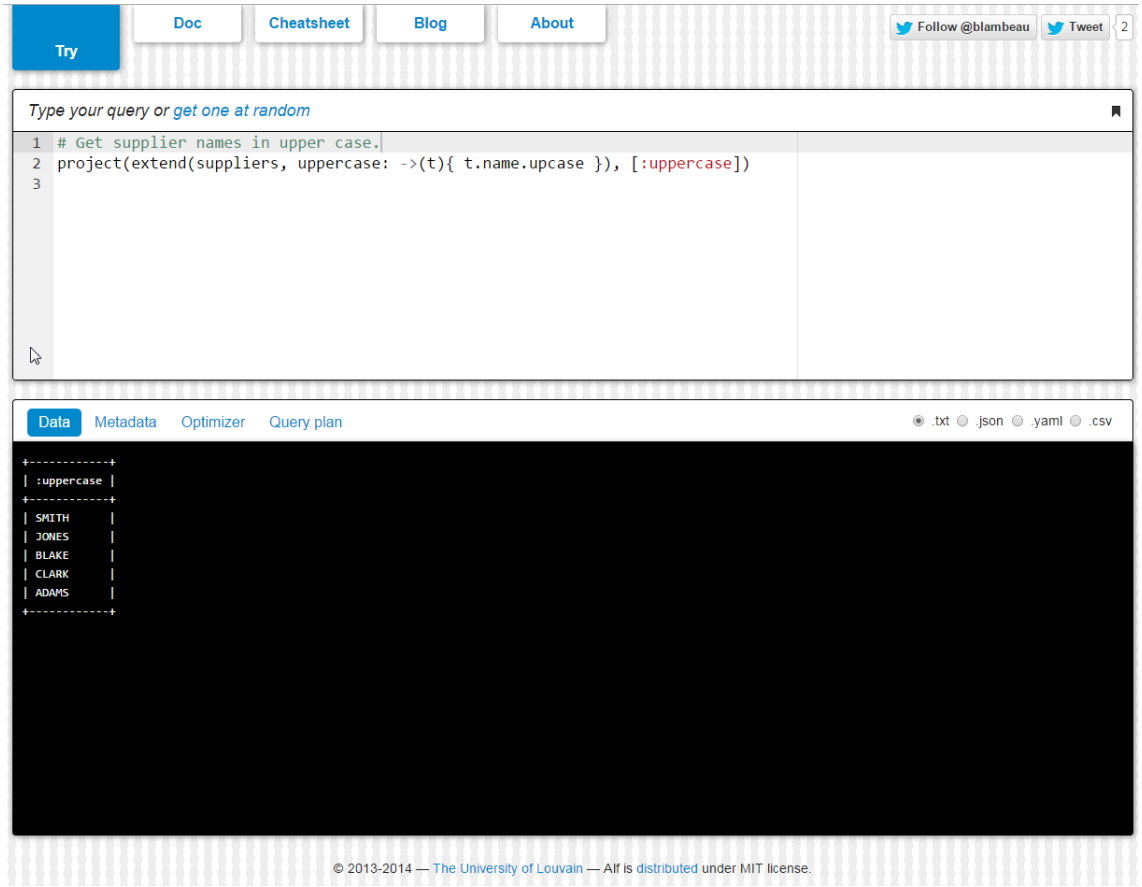
Obrázek 28: Obrazovka testovaného webu Relational Algebra Calculator

Tento program má rozhraní rozdělené na dvě záložky (viz Obrázek 28, kde první záložka je určena pro relační algebru a druhá pro jazyk SQL. Tyto záložky jsou svázány a navzájem se aktualizují jak uživatel píše příkazy. Náš program nepodporuje příkazy v jazyce SQL.

V záložce pro relační algebru je možné příkaz zadávat dvěma způsoby. Buď přímo řec-

kými znaky, které se vkládají pomocí tlačítek nebo názvy daných znaků (pi, sigma, atd.). Námi vytvořený programu používá specifická klíčová slova.
Samotná záložka je rozdělená na tři části. První částí je okno pro textový vstup s potřebnými tlačítky pro vkládání znaků a spouštění dotazu. Druhá část slouží pro zobrazení operátorového stromu a v poslední části se nacházejí data. V našem programu najdeme stejné tři části, avšak program RAC dokáže zobrazit data v jednotlivých uzlech stromu. Náš program na rozdíl od toho zobrazuje stále data v kořenovém uzlu.
Ihned po načtení stránky se dá pracovat s přednastavenými daty. Tuto vlastnost náš program zvládá také. Stejně jako v našem programu je možnost si vytvořit vlastní zdroje dat, nicméně implementovaný způsob importování tabulek ze souboru CSV je velmi jednoduchý. Data v RAC si uživatel může vytvářet rovnou třemi způsoby. První možnost je vytvořením tabulky pomocí příkazů v textovém poli. Druhý způsob je zkopírování SQL skriptu (SQL-dump). Poslední možností je vytvoření grafické tabulky – komponenty, do které uživatel nastaví data. Uživatel má na výběr více druhů přednastavených dat (např. data z anglické stránky o relační algebře na serveru Wikipedia).

7.4.2 Alf – Relational Algebra



Obrázek 29: Obrazovka testovaného webu ALF

Na Obrázku č.29 jde vidět rozložení testovaného webu. Rozdělený je do pěti záložek. První záložka obsahuje samotný procesor. Na druhé záložce můžeme najít dokumentaci k programu. Na třetí záložce se nachází seznam použitelných operátorů. Na posledních dvou záložkách se nachází blog autora a informace o programu.

Procesor je rozdělený na 2 části. První je textové pole pro zadávání dotazů. Druhá část je okno zobrazující výsledky. Nad textovým polem se nachází odkaz, který vytvoří náhodný dotaz. Dotaz je na první pohled poměrně složitý. Zobrazení dat ve spodní části je rozděleno na čtyři záložky. Základní zobrazení je v záložce data. Dále se zde nachází záložky Metadata, Optimizer a Query plan. V záložce *Query plan* se nachází dotaz přepsaný do jazyka SQL. V záložce *Optimizer* je zobrazený uživatelem zadaný dotaz a zároveň jeho

optimalizovaná forma. Dále se v části s výsledkem nachází přepínač, kde vybíráme v jakém formátu chceme data zobrazit. Uživatel má na výběr zobrazení dat jako text, csv formát, yaml a data serializovaná pro json. Na rozdíl od našeho programu zde není operátorový strom ani zobrazení tabulek s daty, případně možnost jak data importovat.

8 Závěr

V této práci jsme se zabývali relační algebrou a jejími operátory. Vytvořili jsme procesor relační algebry s grafickým uživatelským prostředím. Pro implementaci jsme použili různé postupy. Za připomínku stojí například implementace operátorů join pomocí metody nested loops nebo implementace provádění příkazu pomocí operátorového stromu. Také jsme program otestovali a porovnali s dalšími procesory. Práce může po zavedení optimalizací sloužit jako základ pro procesor jazyka SQL nad relační databází.

Ze začátku se vytvoření procesoru jevilo velmi náročné, ale postupným rozšiřováním programu jsem si uvědomil spoustu užitečných informací. Jako příklad můžu zmínit jak probíhá návrh jazyka a jeho gramatika, což jsem nikdy předtím neslyšel. Dále jak se implementují operátory pro práci s databázovými strukturami nebo jak se pracuje s komponentou treeView.

Na konci práce jsem program otestoval s různým množstvím dat a testy podle mého názoru dopadly dobře. Jde z nich vidět, že program je vhodný pro menší počty záznamů, ale je schopný v průběhu pár desítek vteřin zpracovat i tabulky se sto tisíci záznamy. Myslím si, že po důkladných úpravách a optimalizacích by bylo možné jej používat jako alternativu pro přístup k databázi. Při porovnání s dalšími podobnými programy jsem došel k závěru, že vytvořený program je na horší úrovni.

Práce pro mne byla velmi přínosná a doufám, že tyto zkušenosti budu nadále využívat.

Jiří Buchlovský

9 Reference

[1] Algebra. *Wikipedia* [online]. 1. 7. 2014 [cit. 2015-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/Algebra

[2] CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM* [online]. vol. 26, issue 1, s. 64-69 [cit. 2015-05-02]. DOI: 10.1145/357980.358007. Dostupné z: http://portal.acm.org/citation.cfm?doid=357980.358007

[3] *SQLite* [online]. 2015 [cit. 2015-05-05]. Dostupné z: https://www.sqlite.org/

[4] Relational algebra calculator 0.13. *Relational algebra calculator* [online]. 2014 [cit. 2015-05-03]. Dostupné z: http://138.232.66.66/ra/calc.htm

[5] Alf Relational Algebra - Try! [online]. 2014. [cit. 2015-05-05]. Dostupné z: http://www.try-alf.org/

A Příloha na CD/DVD

Na přiloženém CD se nachází adresář s projektem a CSV soubor, ve kterém jsou přischystaná testovací data.